Investigating Termination of Affine Loops With JPF

Kevin Durant
Stellenbosch University

Willem Visser
Stellenbosch University

Corina S. Păsăreanu
NASA Ames Research Center

Abstract

We present some preliminary work on how to discover infinite paths through while loop programs in which the variables in the loop condition are only transformed with affine functions. The infinite paths gathered in this manner are repetitive, that is, after a fixed number of iterations the loop condition is no nearer to being violated than it was initially. A proof is given that shows that this period of repetition is 2 for the one variable case, while for the two variable case simulations suggest that the maximum period is at least 6, but a fixed period is not yet known. The algorithm is implemented as a listener in Symbolic Java PathFinder, and this implementation formed part of the Google Summer of Code 2011.

1. Introduction

An affine loop program, or linear loop program elsewhere in the literature, is a while loop consisting of only affine transformations within the loop body. We can define an affine loop program P as follows:

```
while ($\boldsymbol{y} > \boldsymbol{0}$) {
    $\boldsymbol{x} = A\boldsymbol{x} + \boldsymbol{c}$;
}
```

The guard constraint of such a loop is in general more complex than in the structure presented above, but this simple form befits the presentation of our algorithm, and, to an extent, more complicated constraints should not cause too much of a problem. In the above loop, $\boldsymbol{x}$ is a column vector of the program variables of P, $\boldsymbol{y}$ consists of a subset of these variables, and A and $\boldsymbol{c}$ are an integer matrix and a constant vector respectively. Throughout this text, the program variables $\boldsymbol{x}$ will be restricted to integer values.

Because this form of loop appears often in program code, its behavioural correctness is, in defiance of its technical simplicity, rather important. Termination forms a critical aspect of this correctness, and thus, given an affine loop, we wish to gain some insight into its termination properties.

2. Related Work

There currently exist both theoretical and practical results for this form of loop. Significant results on the decidability of their termination properties are presented by Tiwari [1] and Braverman [2]. In practice, the standard procedure for termination proving of loop programs is the construction, or synthesis, of ranking functions. A welcoming primer to the process is found in a recent paper by Cook [3]. For further reading, one can refer to [4]–[8].

Our technique differs from the standard procedure slightly in that, instead of attempting to synthesise a ranking function which proves the termination of a loop, we search for infinite paths inside the loop program, thus proving the (possible) *non*-termination of the program. The algorithm which we present is still a work-in-progress, and as such there is both theoretical and practical ground yet to be covered. Fortunately, however, the algorithm has been implemented and has already afforded us some useful results, even at this early stage.

3. The Algorithm

The execution paths which are of interest to us are those which are periodic with regard to their effect on the variables found within the loop guard; that is, the behaviour of the path is repetitive (but not necessarily cyclic), and can thus be described by some finite path π. For this form of program, π describes a certain number of iterations of the loop.

Consider the following simple example:

```
while (x > 0) {
    x = -x + 10;
}
```

Figure 1. A potentially infinite affine loop program.

If the input value x is non-positive or greater than 9, the loop program terminates. For values of x in $(1, 9)$, the program is non-terminating, and the sequence of values formed by updating x describes a cycle of length 2. Hence in this case, the behaviour of the path can be fully encapsulated in two iterations of the loop, as is seen when one combines two applications of $f(x)$, the update expression: $f^2(x) = -(-x+10)+10 = x$. Input values in the set $(1, 9)$ produce infinite paths because they do not cause the value of x to oscillate to a non-positive value after the first iteration (i.e., $f(x) > 0\ \forall x \in (1, 9)$).

Applying our algorithm to the above loop produces the following output:

```
=======================
Begin Search (Depth: 4)
=======================

[1]  Not Recurrent: x[2]  <  x[1]
     for (6)

[2]  Recurrent: try (1)

[3]  Not recurrent: x[6]  <  x[3]
     for (1)

[4]  Recurrent: try (1)

=======================
End Search
=======================
```

The second result (`[2] Recurrent`), tells us that there is a periodic path of length 2 which does not terminate, and provides the example input value of $x = 1$. A periodic path of length 4 which fails to terminate will thus also exist (two traversals of the path of length 2 generates an infinite periodic path of length 4), and is detected by the algorithm.

We seek finite paths π which execute in such a way as to leave the loop guard variables no nearer to violating the guard condition than they were before execution, without causing them to violate this condition at any point of the execution of π. If such a π maintains this property when executed repeatedly, we can construct a non-terminating path through the loop. We can automatically locate paths which express this property at least once by altering the loop slightly, making use of a technique first suggested by [9] and also discussed in [3]:

```
picked = false;
while (x > 0) {
    if (*) {
        x0 = x;
        picked = true;
    }
    else if (* && picked) {
        assert(x0 > x);
    }
    x = ax + c;
}
```

Figure 2. A modified single-variable affine loop.

Here `x0`, or x_0, is used to mark the value of x before a certain number of loop iterations are applied. The `(*)` symbols represent non-deterministic choices, and thus allow the before and after values of the variable to be compared over paths containing any number of iterations. The above example specifically displays the case where there is only one program variable and the loop guard is of the form $(x > 0)$. Additional variables require extra assertions; a simple heuristic [10] is used to decide upon the assertion (or ranking) condition:

Guard	Assert
$x > 0$	$x_0 > x$
$x < 0$	$x_0 < x$
$x \neq 0$	$\lvert x_0 \rvert > \lvert x \rvert$

Executing Java Pathfinder[1] on the code with the aid of *jpf-symbc*[2], its symbolic execution extension, will generate execution cases which cause the assertion to be violated. The information we extract from such a case is a path π and an initial value x_0, such that when π travels from x_0 to an end value $f_\pi(x_0)$ (where $f_\pi(x)$ denotes the compound effect of the updates applied by π on the variable x), it exhibits the desired property that $x_0 \leq f_\pi(x_0)$, i.e., the value of the variable is no nearer to violating the loop guard after π has been traversed. For figure 1, a possible π could denote the execution of two loop iterations, with, say, $x_0 = 2$ and $f_\pi(x_0) = f^2(x_0) = -(-2 + 10) + 10 = 2$.

Note that in the broader sense, where the loop forms only a section of a larger program, there may be additional restrictions which are applied to the variables before the loop begins. It is in this scenario that the stem of executions π_0 which leads to the path π must also be taken into account, as it proves that the relevant starting value can be reached.

1. http://babelfish.arc.nasa.gov/trac/jpf
2. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

Once a periodic path π for which $x_0 \leq f_\pi(x_0)$ has been found, we check whether this behaviour will be repeated with subsequent traversals of π. That is, whether $f_\pi^n(x_0) \leq f_\pi^{n+1}(x_0)$ for all $n \geq 0$.

This check is done more abstractly, as we define the candidate set q_π for a given path π to be $(x \mid x > 0 \wedge x \leq f_\pi(x))$. If we can show that:

1) $q_\pi \neq \emptyset$, and
2) $x \in q_\pi \Rightarrow f_\pi(x) \in q_\pi$,

then we call q_π a **recurrent set**. The second condition is equivalent to $(x > 0 \wedge x \leq f_\pi(x) \Rightarrow f_\pi(x) \leq f_\pi^2(x))$, and implies the retention of this non-decreasing property. In the case that such a recurrent set exists, π may indicate an infinite periodic path.

This method may appear incomplete if one considers the possibility of infinite paths which do not exhibit periodic behaviour. Fortunately, however, we can show its completeness for at least the simplest case:

Theorem 1: If a single-variable affine loop program P with a simple loop guard of the form $(x > 0)$ is non-terminating then $\exists \pi_0$ and π such that q_π defines a recurrent set.

Proof: Assume that P is non-terminating, so an infinite path inside P exists. This path must perform infinitely many iterations of P, applying the update expression $f(x) = ax + c$ each time. Somewhere along this path there must exist a pair of consecutive iterations which does not leave its input value x_k decreased, that is, $x_k \leq f^2(x_k)$. If this was not the case, the value of x would have decreased after every second iteration, and would eventually become negative. We can let π denote the execution of two loop iterations and build the set q_π, with $f_\pi = f^2$. This set is not empty, as $x_k \in q_\pi$. We also know that $f^2(x) = a^2x + c(a+1)$, a non-decreasing function, so $x \leq f^2(x) \Rightarrow f^2(x) \leq f^4(x)$. This implies that q_π is recurrent. We thus have a stem π_0 from the initial value to x_k, an iteration pair π and a set q_π for which the recurrent property $x \in q_\pi \Rightarrow f_\pi(x) \in q_\pi$ holds. $\square$

This theorem suggests that there may be an upper bound on the period of the path π: in the single-variable case, this value is 2, and for the two-variable case simulations suggest a value of 6. We still remain with the open question: for an affine transformation in k variables, does an integer n exist such that f^n is always a non-decreasing function?

In the previous proof, the path was assumed to be infinite, and thus the value of x was never negative. When investigating a path for a recurrent set, however, we must not only check that the value of the variable has not been decreased by the traversal of π, but also that it does not fall below 0 at any of the relevant states. We thus strengthen our definition of the candidate set q_π to include this positivity check: if $f_\pi = f^k$, and $f^0(x) = x$, then $q_\pi = (x \mid x > 0 \wedge \cdots \wedge f^{2k-1}(x) > 0 \wedge x \leq f^k(x))$. We now call q_π recurrent if:

1) $q_\pi \neq \emptyset$,
2) $(f^i(x) > 0 \, \forall i \in \{0, \ldots, 2k-1\} \wedge x \leq f^k(x)) \Rightarrow (f^k(x) \leq f^{2k}(x))$.

It is this check which has been used while implementing the algorithm.

4. Implementation

Our current implementation considers an affine loop program P exclusively, separate from the rest of the program in which it may be contained. The algorithm is implemented as a JPF listener, **AffineLoopListener**, and searches for periodic paths π according to the number of iterations applied by the path, up to and including a user-defined limit N.

To accomplish this, the user must supply a Java method containing the affine loop. The loop variables must be flagged as symbolic, so that their update expressions can easily be detected by the listener. The listener executes the first iteration of the loop to allow the symbolic expressions for these updates to be built, and, once complete, fetches these expressions from the stack attributes associated with each symbolic integer, before returning the JPF virtual machine to its former state. Each loop variable is then treated as an object in order to simplify access to the relevant data — symbolic integers, guard conditions, virtual machine indices and compound update expressions — associated with it.

For example, a simple double-variable while loop could be presented to the listener as follows:

```
void twoVariable (int x, int y) {
    while (x > 0) {
        x = x + y + 2;
        y = -x;
    }
}
```

Figure 3. An example loop containing two simultaneous variable updates.

The relevant **.jpf** configuration would then need to declare the `twoVariable` method as symbolic, with two symbolic arguments. Upon execution, the listener reports the fetched update expressions in symbolic terms:

```
x = (y_2_SYMINT + x_1_SYMINT) + CONST_2
y = (CONST_0 - x_1_SYMINT)
```

Note how the update expressions are, in this case, interpreted as simultaneous. This behaviour has been enabled here to provide an analogy to the matrix form of affine loops, and can easily be disabled.

The listener then searches for recurrent sets generated by paths of length j, where j iterates over $\{1, \ldots, N\}$. This enumerative approach to the building of q_π is a generalisation of the path search presented in figure 2, and is suited to the case where the loop program is separate from a parent program. As stated earlier, the recurrence check takes on the following form:

1) $(x \mid x > 0 \land \cdots \land f^{2j-1}(x) > 0 \land x \leq f^j(x)) \neq \emptyset$,
2) $(x > 0 \land \cdots \land f^{2j-1}(x) > 0 \land x \leq f^j(x)) \Rightarrow (f^j(x) \leq f^{2j}(x))$.

Because the symbolic execution library is built to prove satisfiability, condition 2) must be transformed into a solvable form:

2) $(x \mid x > 0 \land \cdots \land f^{2j-1}(x) > 0 \land x \leq f^j(x)$ $\land f^j(x) > f^{2j}(x)) = \emptyset$.

If the solver is unable to find an x which satisfies the above condition, the previous form has been proven valid. If, however, the condition is satisfied, the set being checked is not recurrent. The above condition is stored as two separate constraints by the listener, because the first, $(x > 0 \land \cdots \land f^{2j-1}(x) > 0 \land x \leq f^j(x))$, describes q_π, and can thus be used to check the validity of condition 1). In addition, if condition 2) holds (i.e., the set is empty), solving the first constraint will produce example input values which generate infinite paths.

When multiple variables are present in the guard condition of the loop, the two recurrence conditions will effectively be duplicated for each variable's update expression.

Each compound update expression (those of the form $f^k(x)$) must be computed symbolically by applying the update expression repeatedly, and this causes the stored symbolic expressions to become complicated quite rapidly. As can be seen, when checking a set for a path of length k, the update expressions must be calculated as far as $2k$. Currently, it is the checking of these large expressions which requires the majority of the necessary computation. When multiple variables are referenced by different expressions, the expressions grow at an even quicker rate.

5. Example Results

When applied to a loop, the listener reports the results according to the lengths of the tested paths. Specifically, if the first condition cannot be satisfied for a certain path length, no paths with the desired periodic property exist, so the listener abandons the current check and continues to check paths of a greater length. If only the first condition is satisfied, the listener produces the case which caused the set to fail the second condition. Lastly, when both conditions hold, an example case is produced.

Thus, for the example loop presented in figure 3, the output of a search where paths are limited to a maximum of 6 iterations would be as follows:

```
======================
Begin Search (Depth: 6)
======================

[1]  Not recurrent: x[2]  <  x[1]
     for (3,-2)

[2]  Not recurrent: x[4]  <  x[2]
     for (1,-1)

[3]  Not recurrent: x[6]  <  x[3]
     for (1,-2)

[4]  Not recurrent: x[8]  <  x[4]
     for (1,-2)

[5]  Not recurrent: x[10]  <  x[5]
     for (3,-3)

[6]  Recurrent: try (1,-2)

======================
End Search
======================
```

Here `x[k]` denotes $f^k(x)$, the value of the x variable after k transformations. The recurrence check for paths which span two iterations failed because, for $(x_0, y_0) = (1, -1)$, we have

$$\begin{aligned} f^i(x_0) > 0 \ \forall i \in \{1,2,3\}, &\text{and} \\ x_0 = 1 \leq 3 = f^2(x_0), &\text{but} \\ f^2(x_0) = 3 > 2 = f^4(x_0), & \end{aligned}$$

and a counter-example for the validity of condition 2) is found by the solver. This specific example is recurrent over a minimum of 6 iterations, as the loop transformation forms a cycle of length 6, shown here on input $(1,-2)$:

$$\begin{aligned}&(1,-2)\rightarrow(1,-1)\rightarrow(2,-1)\\ \rightarrow&(3,-2)\rightarrow(3,-3)\rightarrow(2,-3)\\ \rightarrow&(1,-2).\end{aligned}$$

We can also consider a case where multiple variables must remain positive:

```
while (x > 0 && y > 0) {
    x = x + y;
    y = x - y;
}
```

Figure 4. An example loop with two positive variables.

and, limited to 2 iterations, obtain the following results:

```
=====================
Begin Search (Depth: 6)
=====================

[1]  Not recurrent: y[2]  <  y[1]
     for (4,1)

[2]  Recurrent: try (2,1)

=====================
End Search
=====================
```

6. Conclusion

In terms of efficiency, the current implementation can handle only small programs, as much of the practical work which remains involves optimising the repeated application of symbolic update expressions and the simplification of their storage. In addition, the efficiency of the algorithm is directly linked to that of the symbolic solver library used.

To apply the algorithm more practically, the implementation must be expanded to include extra preconditions which may be presented by initialisation code outside of the loop program. The results produced by the algorithm can then be layered onto these preconditions to create valid example values. In a similar vein, the algorithm can be expanded to not only provide example values for infinite paths, but to supply the set of all values which result in non-termination.

Lastly, the completeness of the algorithm is only partly understood, and remains to be investigated at a more fundamental, mathematical level. In particular, how does theorem 1 generalise (or fail to generalise) to higher dimensions?

Furthermore, although the work is of interest theoretically, the extent of its practical impact is still questionable. This depends largely on the prevalence of loops which exhibit affine transformations on variables.

7. Acknowledgements

We would like to thank Andreas Podelski for his valuable insights during the early discussions around this work. We would also like to thank Google for their support during the GSoC 2011[3].

References

[1] A. Tiwari, "Termination of linear programs," in *Computer Aided Verification*, 2004, pp. 70–82.

[2] M. Braverman, "Termination of integer linear programs," in *Computer Aided Verification*, 2006, pp. 372–385.

[3] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Commun. ACM*, vol. 54, pp. 88–98, May 2011.

[4] M. Colón and H. Sipma, "Synthesis of linear ranking functions," in *Tools and Algorithms for Construction and Analysis of Systems*, 2001, pp. 67–81.

[5] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Verification, Model Checking and Abstract Interpretation*, 2004, pp. 239–251.

[6] M. Colón and H. Sipma, "Practical methods for proving program termination," in *Computer Aided Verification*, 2002, pp. 442–454.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma, "Termination analysis of integer linear loops," in *International Conference on Concurrency Theory*, 2005, pp. 488–502.

[8] ——, "Termination of polynomial programs," in *Verification, Model Checking and Abstract Interpretation*, 2005, pp. 113–129.

[9] B. Cook, A. Podelski, and A. Rybalchenko, "Termination proofs for systems code," in *Proceedings of PLDI*, 2006.

[10] D. Dams, R. Gerth, and O. Grumberg, "A heuristic for the automatic generation of ranking functions," in *Workshop on Advances in Verification*, 2000, pp. 1–8.

3. http://code.google.com/soc/